COMP1100 Assignment 3
Technical Report

Tim James
Lab 14 - Wed (14:00 – 16:00) - Eric Pan
u6947396

1. Introduction

I was tasked to write an AI which could examine a given state of a game of Sushi Go and decide on the optimal move to take. Sushi Go is a game in which long term planning will beat any greedy decision making, so the use of game trees and a minimax algorithm are key to writing a proficient AI.

I aimed to take an iterative approach, with each AI I develop increasing in complexity and proficiency, and keeping each previous iteration for benchmarking purposes. This allows for a step-by-step approach and enables me to observe how different AI perform against one another. Large sample sized duels were used for benchmarking, as well as private tournaments against the official AIs.

2. Content

2.1 Program Design

I began by writing a simple greedy AI, which makes a move based on immediate score gain. This greedy AI was quite limited, as it used a simple score-based heuristic, so I began developing a minimax AI which would perform better in the long term. First, I defined the function `gsMoves` which would find all the resulting `gameState`s from playing each possible move of a given `gameState`, utilizing the `playMove` function from the `src\SushiGo.hs` module. This function was later modified to discard any duplicate moves with `removeDuplicates`. Next, the function `buildGSTree` was written to generate a rose tree representing each possible `gameState` at every turn, which used `gsMoves` at every node. This function also only generated the tree until the max depth, or lookahead, had been reached, or the game had finished.

Along with a simple score difference heuristic - `scoreDiffHeuristic` - my program was now ready for the minimax function `scoreDiffValue`, which used player 1 as the maximizer and player 2 as the minimizer. Now, `scoreDiffRank` was used to generate a list of tuples for the possible picks the AI could make, composed of the card and associated value with picking that card. Finally, `bestRank` was used to find which choice would be optimal, depending on the which player the AI was.

This simple minimax AI strongly outperformed the previous greedy AI, though it was slow and was limited by low lookahead values in the early game. I sought to improve the speed of my AI by performing alpha-beta pruning on my game tree, a method which has no risk of information loss. This was implemented at the minimax stage of the algorithm, meaning that parts of the game tree did not need to be calculated. This improved the efficiency of my AI, outperforming the non-pruned minimax AI.

After observing the significant decrease in runtime from alpha-beta pruning, I sought to implement further pruning. This would allow for a higher lookahead value to be used within the time limit, yielding a more optimal result. I began rewriting my `gsMoves` function, now `gsMovesTrimmed`, to utilize a `trimMoves` function which would examine the current `gameState`, and reduce a list of moves. This reduction took part in two stages. First, if there is a move that is likely to be the best move, such as taking a sashimi to prevent the other player from getting a 10-point bonus, reduce the list to only that move. If none of those moves are possible, then remove any clearly bad moves from the list unless there is no other option. This includes moves like taking a nigiri 1 over a nigiri 2, or taking a third tofu and losing the bonus. While this reduction, or trimming, may risk information being lost, it improves runtimes even further. Unlike the previous methods, the strategy taken is subjective.

After observations from playing against my own AI, I noticed that games usually had a large amount of tofu and eels, so it seemed probable that both players would end up with the maximum for those cards. So, taking those cards was not usually important early game, a fact which `trimMoves` makes use of. This also meant that going over the limit of tofu and losing the bonuses could be avoided unless there was no other choice, which could be the deciding factor of many games.

At this point, my AI was still using a simple score difference heuristic, an aspect it was lacking in. A score difference did not account for the long-term potential certain cards had, so I defined a `scorePotentialHeuristic` function. This function would consider what cards were still in play when assigning values to certain cards. For example, it would encourage picking dumplings if

there was a decent number of dumplings in play, and discourage getting a second tofu if there were other tofu in play.

Results from this new heuristic were suboptimal. It had a negative win to loss ratio against an AI with a simple score difference heuristic, losing 62% of the time in a sample space of 100. The weakness of the new heuristic seemed to be in its runtime, with the old heuristic having enough time for a greater lookahead. Given that the increased processing requirement of the potential score heuristic did not outweigh the benefits, I reverted back to the faster score difference heuristic.

A final adjustment was made to my AI, which was to switch to a simple alpha-beta minimax AI there was enough time to reach the bottom of the game tree. In this case, alpha-beta minimax would yield perfect results, so there was no need to prune further and risk losing information.

2.3 Testing

Since most of my functions require a `gameState` input, a set of different `gameState`s specifically engineered to test standard and edge cases were needed. These were divided into three different groups; initial states, mid states and finished states. For each function that required a `gameState` input, at least one of state from each of these groups were used to test edge cases. This is excluding `trimmedMinimaxPickTests`, because this function gives an error on finished `gameState`s, since it should never be used in that scenario. Every function relevant to the default AI was tested.

When it came to testing for optimality, the aforementioned method of duelling AIs against one another was used. The command `:set +s` was used for a rough estimate of efficiency, and large calculations were generally used to allow the difference in runtime to be more easily identified.

3. Reflection

3.1 Challenges

My greatest challenge was trying to improve my heuristic. At first, it was difficult to identify whether my score potential heuristic was an improvement to my overall AI, over a simple score difference heuristic. From the results of a 100-round duel, this did not seem to be the case. Figuring out the problem with the score potential heuristic was difficult, but it was clear that it had an increased runtime. I was unable to bring its speed on par with a simple score difference heuristic, so ultimately I decided not to use it.

3.2 Improvements

Unlike past assignments in this course, this problem has no clear limit to how optimal a given solution can be. Thus, an AI of this sort can be improved seemingly indefinitely. For my AI, further cases for pruning can be added to improve runtimes, and there is a large amount which have not been considered. The heuristic is likely the weakest aspect of my AI, as it takes a fast and simple approach over a particularly intelligent one.

Word Count: 1237